Advanced Java Programming Course

Network programming with socket

By Võ Văn Hải
Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

- Network Addressing
-

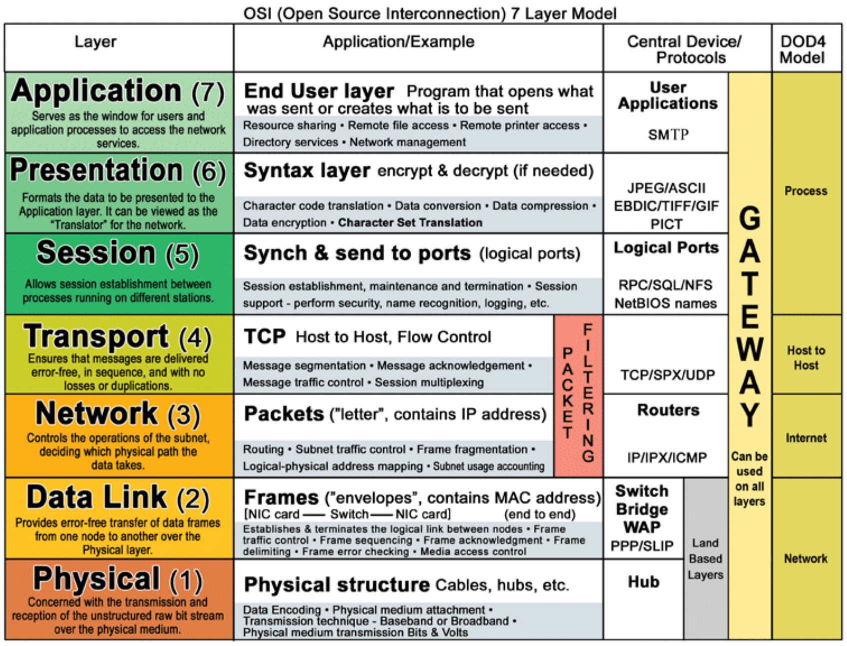
- **Network basics**
  - This is where essential concepts and terms are introduced
- **Using the NetworkInterface class**
  - This provides access to system devices
- **URL/URI/URN**
  - We will discuss how these terms relate to each other
- **The Inet4Address and Inet6Address classes**
  - We will discuss how these are used
- **Network properties**
  - We will consider the properties that are configurable in Java

Understanding network basics

- A network consists of nodes and links that are combined to create network architecture.
- A device connected to the Internet is called a node.
- A computer node is called a host.
- Communication between nodes is conducted along these links using protocols, such as HTTP, or UDP.
- Links can either be wired, such as coaxial cable, twisted pairs, and fier optics, or wireless, such as microwave, cellular, Wi-Fi, or satellite communications

Network architectures and protocols

- Common network architectures include bus, star, and tree-type networks.
  - These physical networks are often used to support an overlay network, which is a virtual network.
  - Such a network abstracts the underlying network to create a network architecture supporting applications, such as peer-to-peer applications.
- When two computers communicate, they use a protocol.
  - There are many different protocols used at various layers of a network. We will mainly focus on HTTP, TCP, IP, and UDP.



OSI (Open Source Interconnection) 7 Layer Model

Layer	Application/Example		Central Device/ Protocols			DOD4 Model
Application (7) Serves as the window for users and application processes to access the network services.	**End User layer** Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management		**User Applications** SMTP		**GATEWAY** Can be used on all layers	Process
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	**Syntax layer** encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation**		JPEG/ASCII EBDIC/TIFF/GIF PICT			
Session (5) Allows session establishment between processes running on different stations.	**Synch & send to ports** (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.		**Logical Ports** RPC/SQL/NFS NetBIOS names			
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	**TCP** Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	PACKET FILTERING	TCP/SPX/UDP			Host to Host
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	**Packets** ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting		**Routers** IP/IPX/ICMP			Internet
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	**Frames** ("envelopes", contains MAC address) [NIC card —— Switch —— NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control		**Switch Bridge WAP** PPP/SLIP	Land Based Layers		Network
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	**Physical structure** Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts		**Hub**			

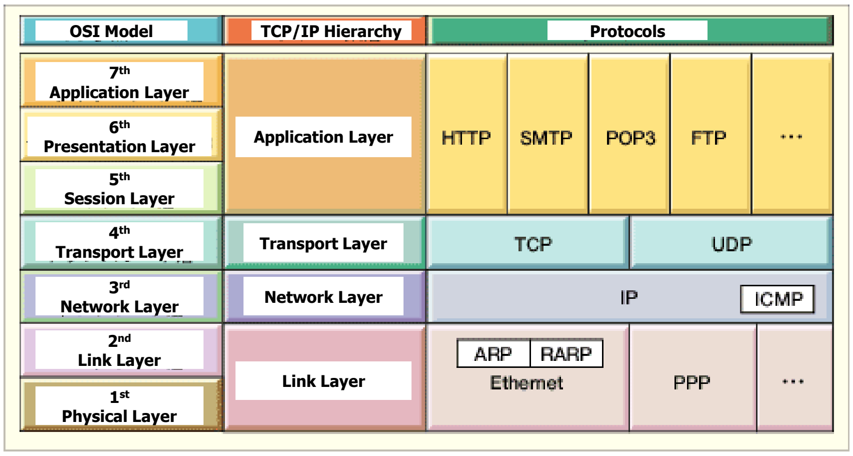
Source: http://buildingautomationmonthly.com/what-is-the-osi-model/

TCP/IP Protocol Suite

- The TCP/IP Model, or Internet Protocol Suite, describes a set of general design guidelines and implementations of specific networking protocols to enable computers to communicate over a network.
- TCP/IP provides end-to-end connectivity specifying how data should be formatted, addressed, transmitted, routed and received at the destination.
- Protocols exist for a variety of different types of communication services between computers.

OSI: Open Systems Interconnection

OSI and Protocol Stack

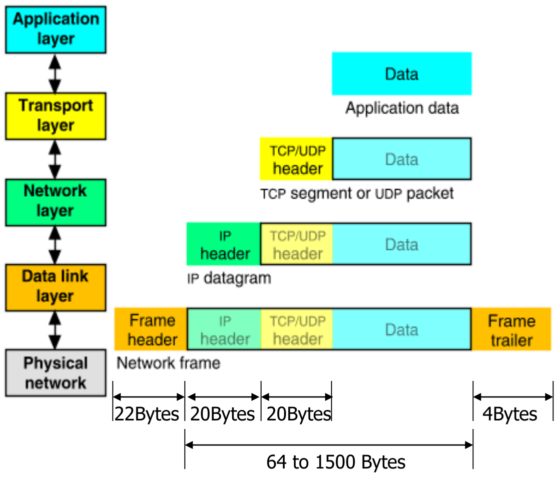


OSI Model	TCP/IP Hierarchy	Protocols				
7th Application Layer	Application Layer	HTTP	SMTP	POP3	FTP	…
6th Presentation Layer						
5th Session Layer						
4th Transport Layer	Transport Layer	TCP		UDP		
3rd Network Layer	Network Layer	IP			ICMP	
2nd Link Layer	Link Layer	ARP RARP Ethernet		PPP		…
1st Physical Layer						

Link Layer : includes device driver and network interface card
Network Layer : handles the movement of packets, i.e. Routing
Transport Layer : provides a reliable flow of data between two hosts
Application Layer : handles the details of the particular application

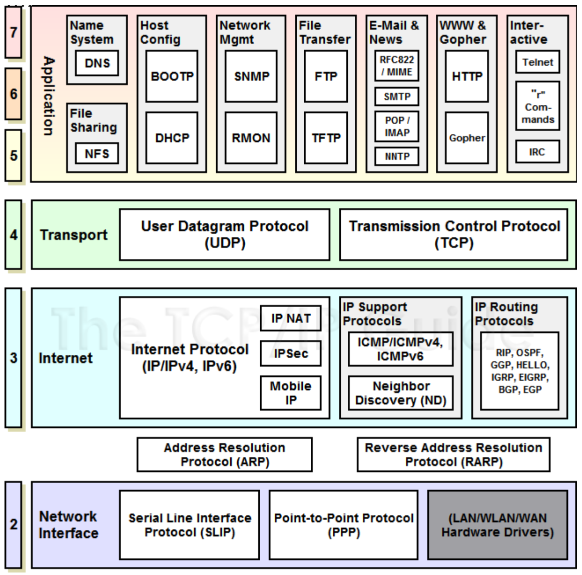
Packet Encapsulation

- The data is sent down the protocol stack
- Each layer adds to the data by prepending headers



9

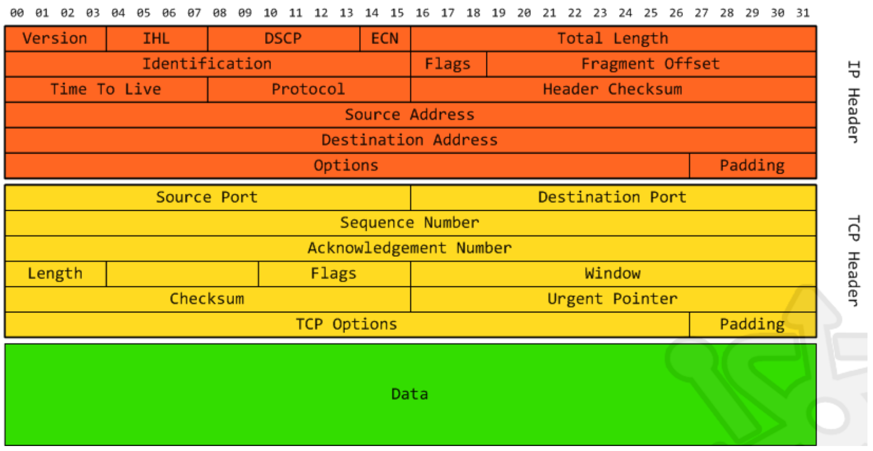
TCP/IP Protocols



10

SRC: http://www.tcpipguide.com/free/t_TCPIPProtocols.htm

IPv4 Packet



Source: http://nullhaus.com/2014/01/deep-packet-inspection-dpi/

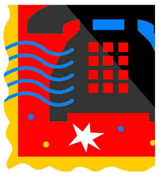
TCP - UDP

- The protocols of the transport layer that we are interested in are TCP and UDP.
- TCP provides a more reliable communication protocol than UDP. However, UDP is bettersuited for short messages when delivery does not need to be robust. Streaming data often uses UDP.

Characteristic	**TCP**	**UDP**
Connection	Connection-oriented	Connectionless
Reliability	Higher	Lower
Order of packets	Order restored	Order potentially lost
Data boundaries	Packets are merged	Packets are distinct
Transmission time	Slower than UDP	Faster than TCP
Error checking	Yes	Yes, but no recovery options
Acknowledgement	Yes	No
Weight	Heavy weight requiring more support	Light weight requiring less support

How Computers Communicate

- When two applications (A and B) on remote machines want to communicate with each other



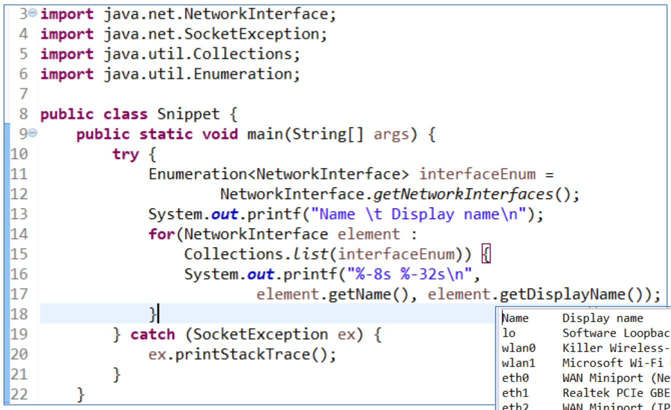
- Using TCP
  - A contacts B and wait for B’s response
  - If B agree, will then respond to A (a connection is established)
  - A & B now can send data back and forth over that connection.

- Using UDP
  - A only needs to know B's address.
  - A send data to B (do not care B is available or not)

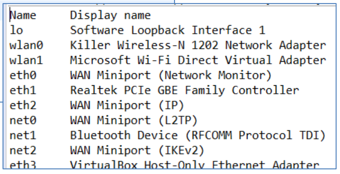
13

Using the NetworkInterface class

The NetworkInterface class provides a means of accessing the devices that act as nodes on a network. This class also provides a means to get low-level device addresses.



```java
import java.net.NetworkInterface;
import java.net.SocketException;
import java.util.Collections;
import java.util.Enumeration;

public class Snippet {
    public static void main(String[] args) {
        try {
            Enumeration<NetworkInterface> interfaceEnum =
                    NetworkInterface.getNetworkInterfaces();
            System.out.printf("Name \t Display name\n");
            for(NetworkInterface element :
                Collections.list(interfaceEnum)) {
                System.out.printf("%-8s %-32s\n",
                        element.getName(), element.getDisplayName());
            }
        } catch (SocketException ex) {
            ex.printStackTrace();
        }
    }
```



```
Name     Display name
lo       Software Loopback Interface 1
wlan0    Killer Wireless-N 1202 Network Adapter
wlan1    Microsoft Wi-Fi Direct Virtual Adapter
eth0     WAN Miniport (Network Monitor)
eth1     Realtek PCIe GBE Family Controller
eth2     WAN Miniport (IP)
net0     WAN Miniport (L2TP)
net1     Bluetooth Device (RFCOMM Protocol TDI)
net2     WAN Miniport (IKEv2)
eth3     VirtualBox Host-Only Ethernet Adapter
```

14

Network addressing concepts

- There are different types of network addresses. An address serves to identify a node in a network.
  - For example, the Internetwork Packet Exchange (**IPX**) protocol was an earlier protocol that was used to access nodes on a network.
  - The X.25 is a protocol suite for Wide Area Network (**WAN**) packet switching.
  - A Media Access Control (**MAC**) address provides a unique identifir for network interfaces at the physical network level.
- However, our primary interests are **IP** addresses.

15

Networking Application Basics
Host & Internet Address

- Host
  - Devices connected to the Internet are called hosts
  - Most hosts are computers, but hosts also include routers, printers, fax machines, soda machines, bat houses, etc.
- Internet addresses
  - Every host on the Internet is identified by a unique, four-byte Internet Protocol (IP) address.
  - This is written in dotted quad format like 199.1.32.90 where each byte is an unsigned integer between 0 and 255.
  - There are about four billion unique IP addresses, but they aren't very efficiently allocated

16

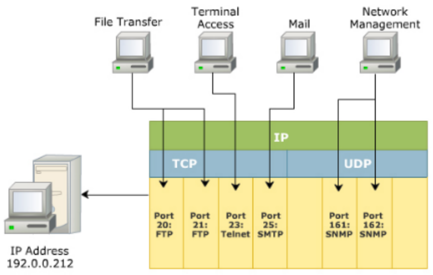
Networking Application Basics
Concept of Ports

- A computer generally has a single physical connection available for the network.
- Several applications running on a machine need to use this single physical connection for communication.
- If data arrives at these physical connections, there is no means to identify the application to which it should be forwarded.
- Hence, data being sent and received on the physical connection are based on the *concept of ports*.

17

Networking Application Basics
Concept of a Ports

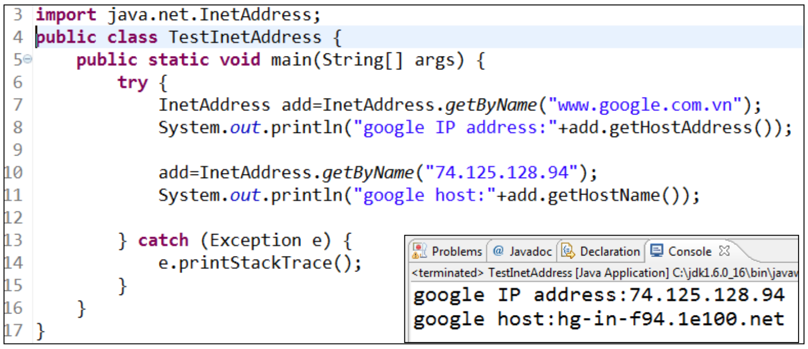
- The physical connection is logically numbered within a range of 0 to 65535 called as Ports.
- The port numbers ranging from 0 to 1023 are reserved
  - for well known services such as HTTP, FTP
  - Your applications should not use any of the port numbers in this range.
- Data transmitted over the Internet is accompanied with the destination address and the port number.
- The destination address identifies the computer
- The port number identifies the application.



18

The InetAddress Class

- The InetAddress represents an Internet Protocol (IP) address
  - It retrieves the host address/name.
  - It provides methods to resolve host names to their IP addresses and vise versa.

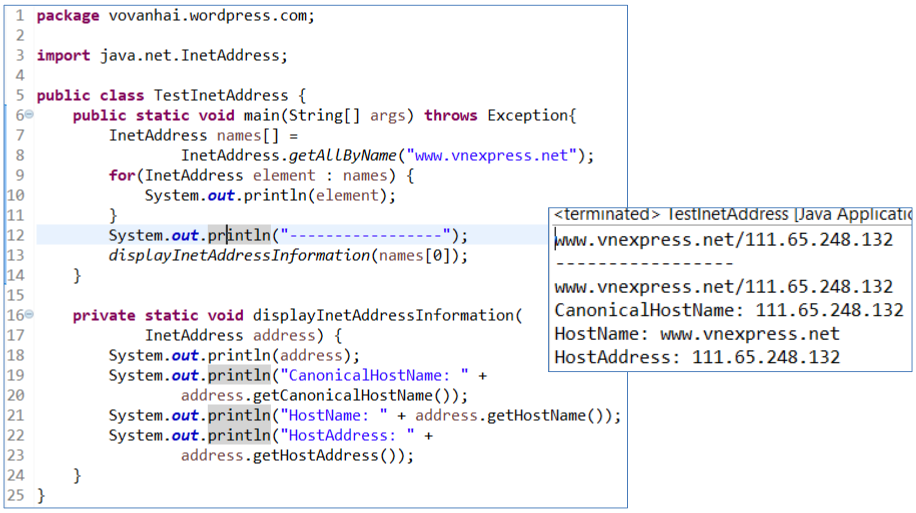


```java
import java.net.InetAddress;
public class TestInetAddress {
    public static void main(String[] args) {
        try {
            InetAddress add=InetAddress.getByName("www.google.com.vn");
            System.out.println("google IP address:"+add.getHostAddress());

            add=InetAddress.getByName("74.125.128.94");
            System.out.println("google host:"+add.getHostName());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
<terminated> TestInetAddress [Java Application] C:\jdk1.6.0_16\bin\javaw
google IP address:74.125.128.94
google host:hg-in-f94.1e100.net
```

19

The InetAddress Class



```java
package vovanhai.wordpress.com;

import java.net.InetAddress;

public class TestInetAddress {
    public static void main(String[] args) throws Exception{
        InetAddress names[] =
                InetAddress.getAllByName("www.vnexpress.net");
        for(InetAddress element : names) {
            System.out.println(element);
        }
        System.out.println("-----------------");
        displayInetAddressInformation(names[0]);
    }

    private static void displayInetAddressInformation(
            InetAddress address) {
        System.out.println(address);
        System.out.println("CanonicalHostName: " +
                address.getCanonicalHostName());
        System.out.println("HostName: " + address.getHostName());
        System.out.println("HostAddress: " +
                address.getHostAddress());
    }
}
```

```
<terminated> TestInetAddress [Java Applicati
www.vnexpress.net/111.65.248.132
-----------------
www.vnexpress.net/111.65.248.132
CanonicalHostName: 111.65.248.132
HostName: www.vnexpress.net
HostAddress: 111.65.248.132
```

20

The Inet6Address class

- IPv6 addresses use 128 bits (16 octets). This permits up to 2^{128} addresses.
- An IPv6 address is written as a series of eight groups, with 4 hexadecimal numbers each, separated by colons.
- The digits are case insensitive. For example, the IPv6 address for www.google.com is as follows:

  2607:f8b0:4002:0c08:0000:0000:0000:0067
- IPv6 supports three addressing types:
  - **Unicast**: This specifies a single network interface.
  - **Anycast**: This type of address is assigned to a group of interfaces. When a packet is sent to this group, only one member of the group receives the packet, often the one that is closest.
  - **Multicast**: This sends a packet to all members of a group.

URL/URI/URN

- An Uniform Resource Identifier (URI) identifies the name of a resource, such as a website, or a file on the Internet. It may contain the name of a resource and its location.
- An Uniform Resource Locator (URL) specifies where a resource is located, and how to retrieve it.
  - Ex: https://www.packtpub.com/ - ftp://speedtest.tele2.net/
- Java provides classes to support URIs and URLs
- A Uniform Resource Name (URN) identifis the resource but not its location.

Read more at
https://en.wikipedia.org/wiki/Uniform_Resource_Identifier#Relationship_between_URI.2C_URL.2C_and_URN

Using the URI class

- The general syntax of a URI consists of a scheme and a scheme-specifi-part:

  [scheme:] scheme-specifi-part

- There are many schemes that are used with a URI, including:
  - **file**: This is used for files systems
  - **ftp**: This is File Transfer Protocol
  - **http**: This is commonly used for websites
  - **mailto**: This is used as part of a mail service
  - **urn**: This is used to identify a resource by name

Using the URI class

Java uses the URI class to represent a URI, and it possesses several methods to extract parts of a URI

```
URI uri = new
        URI("https://www.packtpub.com/books/content/support");
```

Method	Purpose
getAuthority	This is the entity responsible for resolving the URI
getScheme	The scheme used
getSchemeSpecificPart	The scheme specific part of the URI
getHost	The host
getPath	The URI path
getQuery	The query, if any
getFragment	The sub-element being accessed, if used
getUserInfo	User information, if available
normalize	Removes unnecessary "." and ".." from the path

The java.net.URL class

- The URL class represents an URL object.
- The URL class contains methods to
  - create new URLs
  - parse the different parts of a URL
  - get an input stream from a URL so you can read data from a server
  - get content from the server as a Java object
- Supported Protocols

file	ftp	gopher	http	mailto
appletresource	doc	netdoc	systemresource	verbatim

Properties of URL class

Method	**Purpose**
`getProtocol`	This is the name of the protocol.
`getHost`	This is the host name.
`getPort`	This is the port number.
`getDefaultPort`	This is the default port number for the protocol.
`getFile`	This returns the result of `getPath` concatenated with the results of `getQuery`.
`getPath`	This retrieves the path, if any, for the URL.
`getRef`	This is the return name of the URL's reference.
`getQuery`	This returns the query part of the URL if present.
`getUserInfo`	This returns any user information associated with the URL.
`getAuthority`	The authority usually consists of the server host name or IP address. It may include the port number.

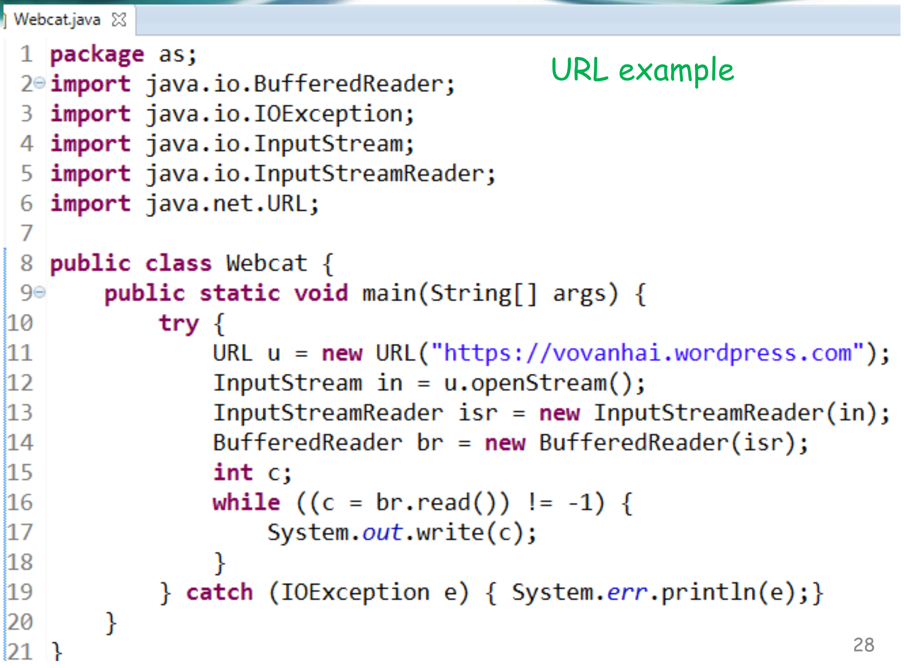
Reading Data from a URL

- The openStream() method connects to the server specified in the URL and returns an InputStream object fed by the data from that connection.

```
public final InputStream openStream() throws IOException
```

- Any headers that precede the actual data are stripped off before the stream is opened.
- Network connections are less reliable and slower than files. Buffer with a BufferedReader or a BufferedInputStream.

URL example



```java
package as;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;

public class Webcat {
    public static void main(String[] args) {
        try {
            URL u = new URL("https://vovanhai.wordpress.com");
            InputStream in = u.openStream();
            InputStreamReader isr = new InputStreamReader(in);
            BufferedReader br = new BufferedReader(isr);
            int c;
            while ((c = br.read()) != -1) {
                System.out.write(c);
            }
        } catch (IOException e) { System.err.println(e);}
    }
}
```

URLConnections

- The java.net.URLConnection class is an abstract class that handles communication with different kinds of servers like ftp servers and web servers.
- Protocol specific subclasses of URLConnection handle different kinds of servers.
- By default, connections to HTTP URLs use the GET method.
- Capacities:
  - Can send output as well as read input
  - Can post data
  - Can read headers from a connection

URLConnection five steps

1. The URL is constructed.
2. The URL's openConnection() method creates the URLConnection object.
3. The parameters for the connection and the request properties that the client sends to the server are set up.
4. The connect() method makes the connection to the server.
5. The response header information is read using getHeaderField().

I/O Across a URLConnection

- Data may be read from the connection in one of two ways
  - raw by using the input stream returned by getInputStream()
  - through a content handler with getContent().
- Data can be sent to the server using the output stream provided by getOutputStream().

```
void doReadSampel()throws Exception{
    URL u = new URL("http://www.wwwac.org/");
    URLConnection uc = u.openConnection();
    uc.connect();
    InputStream in = uc.getInputStream();
    // read the data...
    in.close();
}
```

31

Send request to server

- Since a URLConnection doesn't allow output by default, you have to call setDoOutput(true) before asking for an output stream.

```
public InputStream sendRequest(URL url, String method,
        String params)throws Exception{
    HttpURLConnection con=(HttpURLConnection)url.openConnection();
    con.setRequestMethod(method.toUpperCase());
    con.setDoOutput(true);

    OutputStreamWriter ow=new OutputStreamWriter(con.getOutputStream(),"UTF-8");
    ow.write(params);//send data to server
    ow.flush(); ow.close();

    int rc=con.getResponseCode();//response code from server
    if(rc==HttpURLConnection.HTTP_OK){
        return con.getInputStream();
    }
    return null;
}
```

32

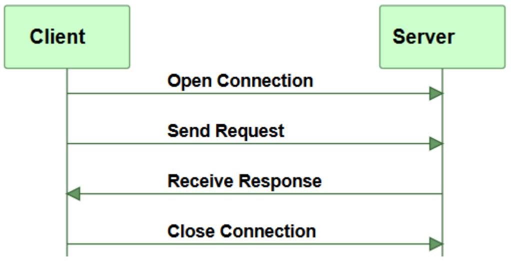
Get response from server

```
public String getResponse(InputStream is)throws Exception{
    BufferedReader br=new BufferedReader(new InputStreamReader(is));
    String line="",html="";
    while((line=br.readLine())!=null){
        html+=line+"\n";
    }
    br.close();
    return html;
}
```

```
public static void main(String[] args) throws Exception{
    SubmitData2Server sr=new SubmitData2Server();

    URL url=new URL("http://localhost:8080/SampleWeb/LogonServlet");
    String params= "us="+URLEncoder.encode("vovanhai", "UTF-8");
    params+="&"+"psw="+URLEncoder.encode("vovanhaiz", "UTF-8");

    InputStream is = sr.sendRequest(url, "POST", params);
    String ret=sr.getResponse(is);

    System.out.println(ret);
}
```

Java TCP Networking Basics

- Typically a client opens a TCP/IP connection to a server. The client then starts to communicate with the server. When the client is finished it closes the connection again.



- A client may send more than one request through an open connection

Using sockets

- A socket is a connection between two hosts. It can perform seven basic operations:
  - Connect to a remote machine
  - Send data
  - Receive data
  - Close a connection
  - Bind to a port
  - Listen for incoming data
  - Accept connections from remote machines on the bound port
- Java's Socket class, which is used by both clients and servers, has methods that correspond to the first four of these operations.

Java Networking: Socket

- In order to connect to a server over the internet (via TCP/IP) in Java, you need to create a java.net.Socket and connect it to the server.
- Creating a Socket

```
Socket socket = new Socket(InetAddress.getByName("localhost") , 80);
```

- Writing to a Socket

```
OutputStream out = socket.getOutputStream();
 out.write( buffer );
```

- Reading from a Socket

```
InputStream in = socket. getInputStream();
 in.read( buffer );
```

- Closing a Socket

```
socket.close();
```

Java Networking: ServerSocket

- In order to implement a Java server that listens for incoming connections from clients via TCP/IP, you need to use a java.net.ServerSocket .
- Creating a ServerSocket

```
ServerSocket serverSocket = new ServerSocket(9000);
```

- Listening For Incoming Connections

```
ServerSocket serverSocket = new ServerSocket(9000);
boolean isStopped = false;
while(!isStopped){
    Socket clientSocket = serverSocket.accept();
    //do something with clientSocket
}
```

- Closing Server Sockets

```
serverSocket.close();
```

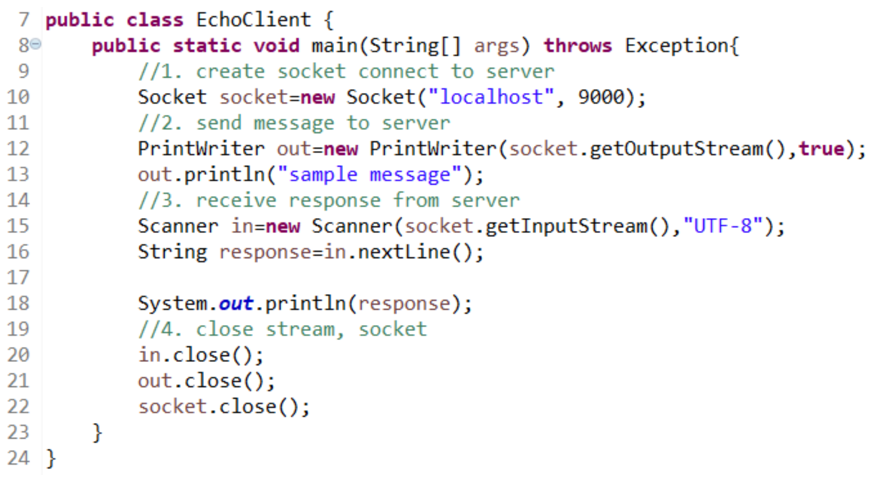
37

Sample server using ServerSocket object

```
public class EchoServer {
    public static void main(String[] args) throws Exception{
        //1. create server-socket listen on port 9000
        ServerSocket serverSocket = new ServerSocket(9000);
        System.out.println("Server is listening on port 9000");
        boolean isStopped = false;
        while(!isStopped){
            //2. accept connect from any client
            Socket clientSocket = serverSocket.accept();
            //3. read the message from client
            InputStream is=clientSocket.getInputStream();
            Scanner in=new Scanner(is,"UTF-8");
            String msg="";
            if(in.hasNextLine())
                msg=in.nextLine();
            //4. processing message
            String response=new StringBuilder(msg).reverse().toString();
            //5. send response to client
            OutputStream os=clientSocket.getOutputStream();
            try(PrintWriter out=new PrintWriter(os,true)){
                out.println(response);
            }
            in.close();
            //6. close client socket
            clientSocket.close();
        }
        serverSocket.close();
    }
}
```

38

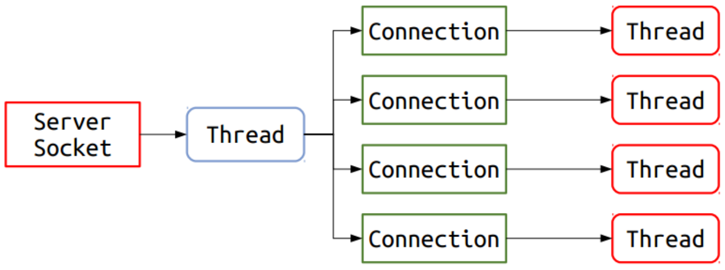
Sample client using Socket object



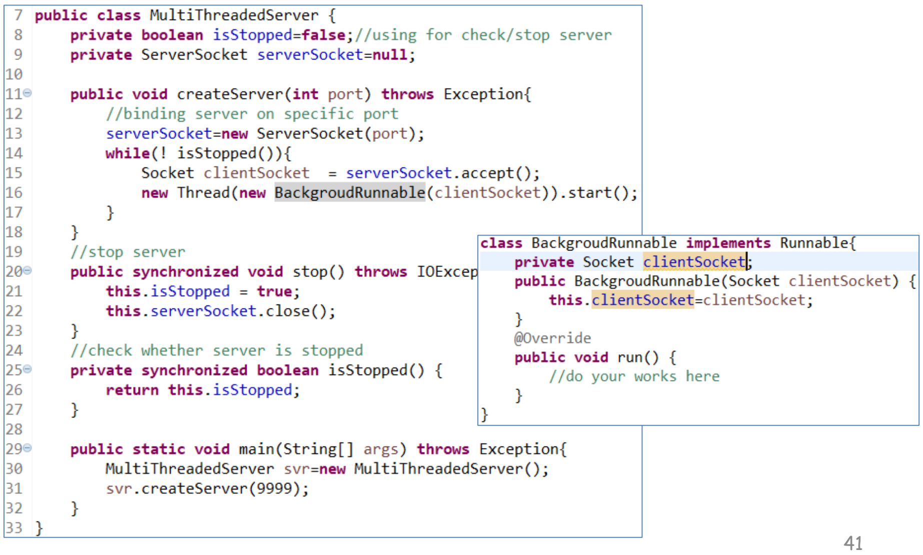
```
public class EchoClient {
    public static void main(String[] args) throws Exception{
        //1. create socket connect to server
        Socket socket=new Socket("localhost", 9000);
        //2. send message to server
        PrintWriter out=new PrintWriter(socket.getOutputStream(),true);
        out.println("sample message");
        //3. receive response from server
        Scanner in=new Scanner(socket.getInputStream(),"UTF-8");
        String response=in.nextLine();

        System.out.println(response);
        //4. close stream, socket
        in.close();
        out.close();
        socket.close();
    }
}
```

Serve multi-clients concurrently

- Create a new process to handle each connection so that multiple clients can be serviced at the same time.
- Java programs should spawn a thread to interact with the client so that the server can be ready to process the next connection sooner



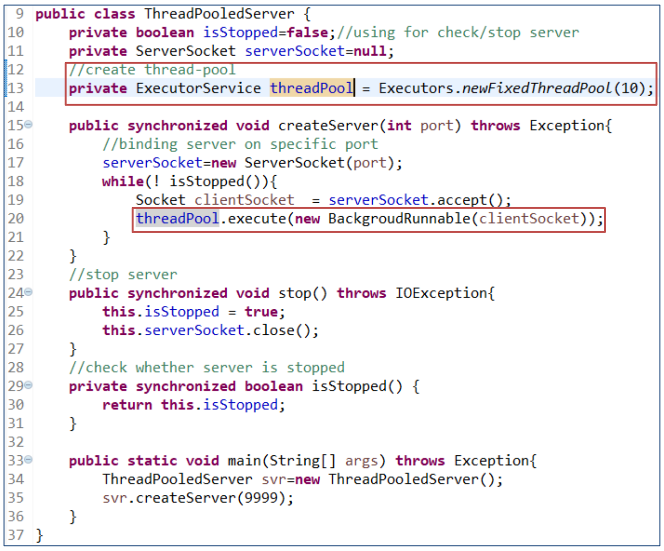
Server multi-clients concurrently



```
public class MultiThreadedServer {
    private boolean isStopped=false;//using for check/stop server
    private ServerSocket serverSocket=null;

    public void createServer(int port) throws Exception{
        //binding server on specific port
        serverSocket=new ServerSocket(port);
        while(! isStopped()){
            Socket clientSocket  = serverSocket.accept();
            new Thread(new BackgroudRunnable(clientSocket)).start();
        }
    }
    //stop server
    public synchronized void stop() throws IOExcep
        this.isStopped = true;
        this.serverSocket.close();
    }
    //check whether server is stopped
    private synchronized boolean isStopped() {
        return this.isStopped;
    }

    public static void main(String[] args) throws Exception{
        MultiThreadedServer svr=new MultiThreadedServer();
        svr.createServer(9999);
    }
}
```

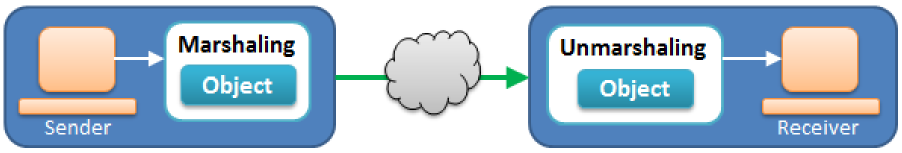
```
class BackgroudRunnable implements Runnable{
    private Socket clientSocket;
    public BackgroudRunnable(Socket clientSocket) {
        this.clientSocket=clientSocket;
    }
    @Override
    public void run() {
        //do your works here
    }
}
```

Thread Pooled Server



```
public class ThreadPooledServer {
    private boolean isStopped=false;//using for check/stop server
    private ServerSocket serverSocket=null;
    //create thread-pool
    private ExecutorService threadPool = Executors.newFixedThreadPool(10);

    public synchronized void createServer(int port) throws Exception{
        //binding server on specific port
        serverSocket=new ServerSocket(port);
        while(! isStopped()){
            Socket clientSocket  = serverSocket.accept();
            threadPool.execute(new BackgroudRunnable(clientSocket));
        }
    }
    //stop server
    public synchronized void stop() throws IOException{
        this.isStopped = true;
        this.serverSocket.close();
    }
    //check whether server is stopped
    private synchronized boolean isStopped() {
        return this.isStopped;
    }

    public static void main(String[] args) throws Exception{
        ThreadPooledServer svr=new ThreadPooledServer();
        svr.createServer(9999);
    }
}
```

Object transmission



- The processes of object transmission are:
  - Marshalling object to be transmitted
  - Send marshalled object over the network
  - Unmarshalling object and the execute
- *Note that object class must implement Serializable interface*
- To transmit *serialized objects*, we would use *ObjectInputStream* and *ObjectOutputStream* instead.

Live Demo

User Datagram Protocol (UDP)

- UDP works a bit differently from TCP. When you send data via TCP you first create a connection. Once the TCP connection is established TCP guarantess that your data arrives at the other end, or it will tell you that an error occurred.
- With UDP you just send packets of data (datagrams) to some IP address on the network. You have no guarantee that the data will arrive. You also have no guarantee about the order which UDP packets arrive in at the receiver. This means that UDP has less protocol overhead (no stream integrity checking) than TCP.
- UDP is appropriate for data transfers where it doesn't matter if a packet is lost in transition.

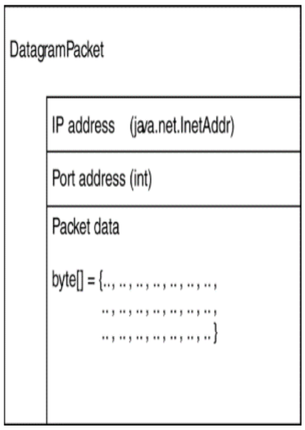
User Datagram Protocol (UDP)

- UDP is still layered ontop of IP. You can use Java's DatagramSocket both for sending and receiving UPD datagrams.
- UDP can be used to send packets.
- The packets can be delivered in random order.
- It is up to the recipient to put the packets in order and to request retransmission of missing packets.
- UDP is most suited for applications where missing packets can be tolerated, for example, in audio or video streams...

User Datagram Protocol (UDP)

The Datagram packet

- An independent, self-contained message sent over the network.
- Used to employ a connectionless packet delivery service.
- Packet delivery is not guaranteed
- Datagram packet structure:
  - Addressing information (IP)
  - Port
  - Sequence of bytes


DatagramPacket
IP address (java.net.InetAddr)
Port address (int)
Packet data
byte[] = {.., ..}

Sending Data via a DatagramSocket

- To send data via Java's DatagramSocket you must first create a DatagramPacket.

```
byte[] buffer = new byte[2048];
DatagramPacket packet = new DatagramPacket(
    buffer,          //the data that is to be sent in the UDP datagram.
    buffer.length,   //the length of the buffer in bytes
    serverAddress,   // the address of the node (eg. server) to send the UDP packet to
    serverPort       //the UDP port the server to receiver the data is listeing on
    );
```

- To send the DatagramPacket you must create a DatagramSocket targeted at sending data.

```
try(DatagramSocket datagramSocket = new DatagramSocket()){
    datagramSocket.send(packet);
}
```

Receiving Data via a DatagramSocket

- Receiving data via a DatagramSocket is done by first creating a DatagramPacket and then receiving data into it via the DatagramSocket's receive() method.

```
try(DatagramSocket datagramSocket = new DatagramSocket(80)){
    //create empty packet
    byte[] buffer = new byte[2048];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

    datagramSocket.receive(packet);
    //packet contains all received information & data
    byte[] data = packet.getData();
    InetAddress fromAddress=packet.getAddress();
    int fromPort=packet.getPort();
}
```

Live Demo

MulticastSocket

- Java uses MulticastSocket class to create UDP multicast sockets to receive datagram packets sent to a multicast IP address.
- A multicast socket is based on a group membership. After creating and bounding a multicast socket, call its joinGroup(InetAddress multiCastIPAddress) method to join the multicast group, any datagram packet sent to that group will be delivered to this socket.
- To leave a group, call the leaveGroup(InetAddress multiCastIPAddress) method

- Java uses MulticastSocket class to create UDP multicast sockets to receive datagram packets sent to a multicast IP address.

Range Start Address	Range End Address	Description
224.0.0.0	224.0.0.255	Reserved for special “well-known” multicast addresses.
224.0.1.0	238.255.255.255	Globally-scoped (Internet-wide) multicast addresses.
239.0.0.0	239.255.255.255	Administratively-scoped (local) multicast addresses.

IP Multicast Address Ranges and Uses

TTL (Time To Live)

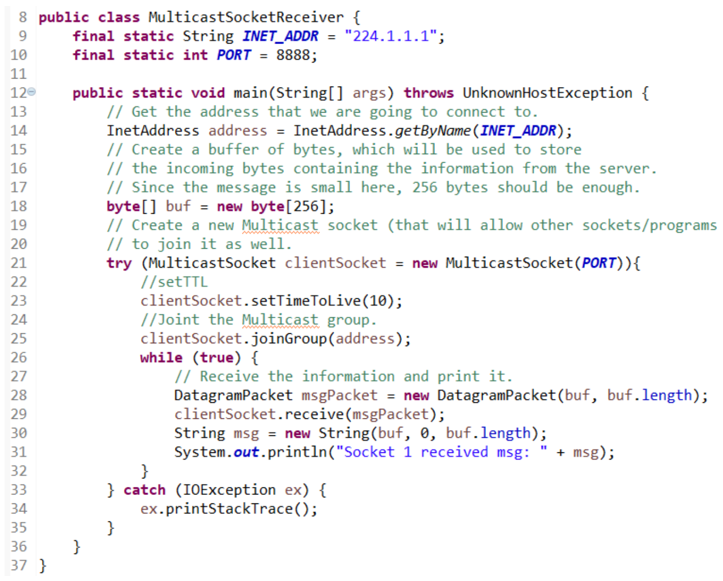
- The TTL (Time To Live) field in the IP header has a double significance in multicast. As always, it controls the live time of the datagram to avoid it being looped forever due to routing errors. Routers decrement the TTL of every datagram as it traverses from one network to another and when its value reaches 0 the packet is dropped.
- The TTL in IPv4 multicasting has also the meaning of "threshold"

TTL (Time To Live)

- A list of TTL thresholds and their associated scope

TTL	Scope
0	Restricted to the same host. Won't be output by any interface.
1	Restricted to the same subnet. Won't be forwarded by a router.
<32	Restricted to the same site, organization or department.
<64	Restricted to the same region.
<128	Restricted to the same continent.
<255	Unrestricted in scope. Global.

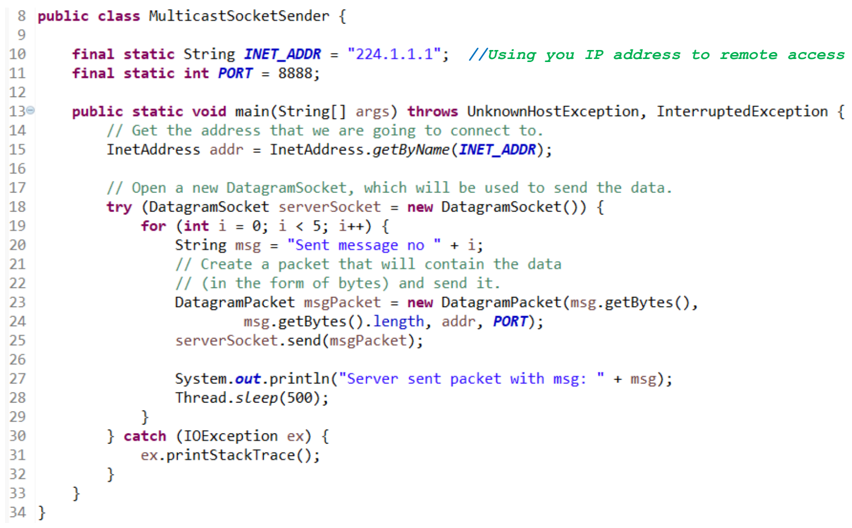
MulticastSocket receiver



```java
public class MulticastSocketReceiver {
    final static String INET_ADDR = "224.1.1.1";
    final static int PORT = 8888;

    public static void main(String[] args) throws UnknownHostException {
        // Get the address that we are going to connect to.
        InetAddress address = InetAddress.getByName(INET_ADDR);
        // Create a buffer of bytes, which will be used to store
        // the incoming bytes containing the information from the server.
        // Since the message is small here, 256 bytes should be enough.
        byte[] buf = new byte[256];
        // Create a new Multicast socket (that will allow other sockets/programs
        // to join it as well.
        try (MulticastSocket clientSocket = new MulticastSocket(PORT)){
            //setTTL
            clientSocket.setTimeToLive(10);
            //Joint the Multicast group.
            clientSocket.joinGroup(address);
            while (true) {
                // Receive the information and print it.
                DatagramPacket msgPacket = new DatagramPacket(buf, buf.length);
                clientSocket.receive(msgPacket);
                String msg = new String(buf, 0, buf.length);
                System.out.println("Socket 1 received msg: " + msg);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

55

MulticastSocket sender



```java
public class MulticastSocketSender {

    final static String INET_ADDR = "224.1.1.1";  //Using you IP address to remote access
    final static int PORT = 8888;

    public static void main(String[] args) throws UnknownHostException, InterruptedException {
        // Get the address that we are going to connect to.
        InetAddress addr = InetAddress.getByName(INET_ADDR);

        // Open a new DatagramSocket, which will be used to send the data.
        try (DatagramSocket serverSocket = new DatagramSocket()) {
            for (int i = 0; i < 5; i++) {
                String msg = "Sent message no " + i;
                // Create a packet that will contain the data
                // (in the form of bytes) and send it.
                DatagramPacket msgPacket = new DatagramPacket(msg.getBytes(),
                        msg.getBytes().length, addr, PORT);
                serverSocket.send(msgPacket);

                System.out.println("Server sent packet with msg: " + msg);
                Thread.sleep(500);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

56

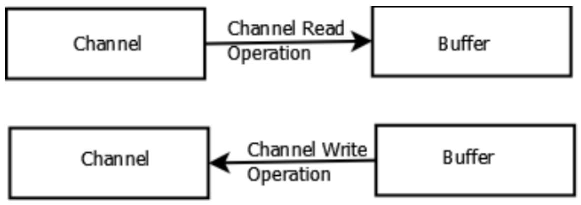
NIO Support for Networking

- NIO is an alternative for the earlier Java IO API and parts of the network API.
- While NIO is a broad and complex topic, our interest is how it provides support for network applications.
- We will explore several topics, including the following:
  - The nature and relationship between buffers, channels, and selectors
  - The use of NIO techniques to build a client/server
  - The process of handling multiple clients
  - Support for asynchronous socket channels
  - Basic buffer operations

57

Java NIO

- Java NIO uses three core classes:
  - Buffer: This holds information that is read or written to a channel
  - Channel: This is a stream-like technique that supports asynchronous read/write operations to a data source/sink
  - Selector: This is a mechanism to handle multiple channels in a single thread



buffers and channels work together to process data

58

Channels & Buffers

Channel class	Purpose
`FileChannel`	This connects to a file
`DatagramChannel`	This supports datagram sockets
`SocketChannel`	This supports streaming sockets
`ServerSocketChannel`	This listens for socket requests
`NetworkChannel`	This supports a network socket
`AsynchronousSocketChannel`	This supports asynchronous streaming sockets

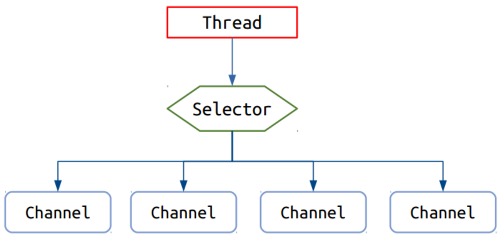
The Selector class is useful when an application uses many low-traffic connections that can be handled using a single thread. This is more effiient than creating a thread for each connection

Buffer class	Data type supported
`ByteBuffer`	`byte`
`CharBuffer`	`char`
`DoubleBuffer`	`double`
`FloatBuffer`	`float`
`IntBuffer`	`int`
`LongBuffer`	`long`
`ShortBuffer`	`short`

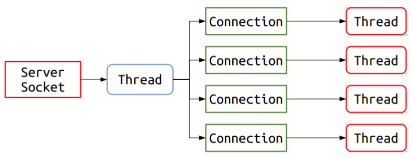
59

Selectors

- A Selector allows a single thread to handle multiple Channel's. This is handy if your application has many connections (Channels) open, but only has low traffic on each connection.



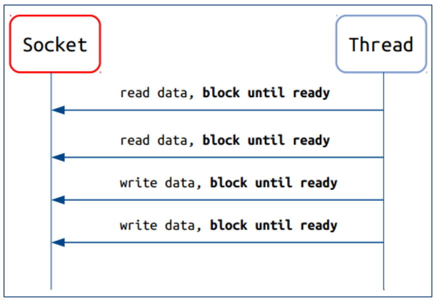
Java NIO server design



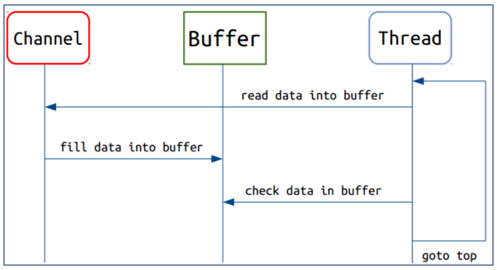
Classic IO server design

60

Java.IO - Blocking I/O


Socket
Thread
read data, block until ready
read data, block until ready
write data, block until ready
write data, block until ready

Java.NIO — Buffer-oriented & Non-blocking I/O



61

Java.IO vs. Java.NIO

Java.IO — Work with data

```
// read from socket
Scanner sc = new Scanner(
        socket.getInputStream());
String string = sc.nextLine();
System. out.println("Received "+string);
// write to socket
PrintWriter pw = new PrintWriter(
        socket.getOutputStream());
pw.println("Hello");

sc.close();pw.close();
```

Java.NIO — Work with data

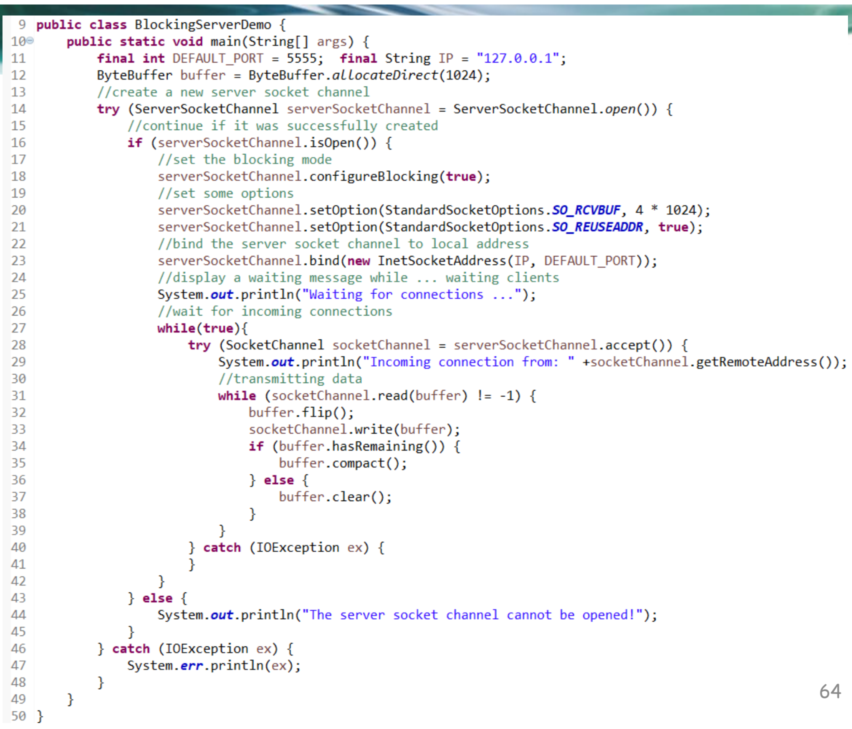
```
ByteBuffer readBuffer = ByteBuffer.allocate(1024);
// prepare to read
readBuffer.clear();
SocketChannel channel = getChannel();
channel.read(readBuffer);
readBuffer.flip();
// reading the buffer
// ...............
ByteBuffer writeBuffer = ByteBuffer.allocate(1024);
// prepare to put data
writeBuffer.clear();
// putting the data
// ...............
// prepare to write
writeBuffer.flip();
channel.write(writeBuffer);
```

62

Writing a Blocking TCP Server

Steps to implement blocking TCP Server

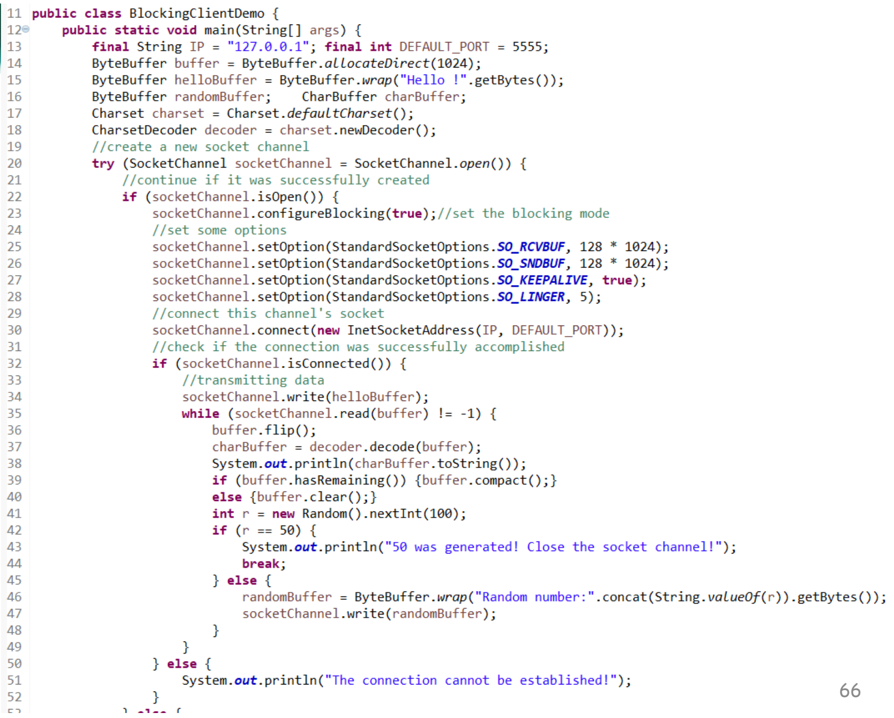
1. Creating a New Server Socket Channel
2. Configuring Blocking Mechanisms
3. Setting Server Socket Channel Options (optional)
4. Binding the Server Socket Channel
5. Accepting Connections
6. Transmitting Data over a Connection
7. Closing the Channel



```java
public class BlockingServerDemo {
    public static void main(String[] args) {
        final int DEFAULT_PORT = 5555;  final String IP = "127.0.0.1";
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
        //create a new server socket channel
        try (ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {
            //continue if it was successfully created
            if (serverSocketChannel.isOpen()) {
                //set the blocking mode
                serverSocketChannel.configureBlocking(true);
                //set some options
                serverSocketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
                serverSocketChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
                //bind the server socket channel to local address
                serverSocketChannel.bind(new InetSocketAddress(IP, DEFAULT_PORT));
                //display a waiting message while ... waiting clients
                System.out.println("Waiting for connections ...");
                //wait for incoming connections
                while(true){
                    try (SocketChannel socketChannel = serverSocketChannel.accept()) {
                        System.out.println("Incoming connection from: " +socketChannel.getRemoteAddress());
                        //transmitting data
                        while (socketChannel.read(buffer) != -1) {
                            buffer.flip();
                            socketChannel.write(buffer);
                            if (buffer.hasRemaining()) {
                                buffer.compact();
                            } else {
                                buffer.clear();
                            }
                        }
                    } catch (IOException ex) {
                    }
                }
            } else {
                System.out.println("The server socket channel cannot be opened!");
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

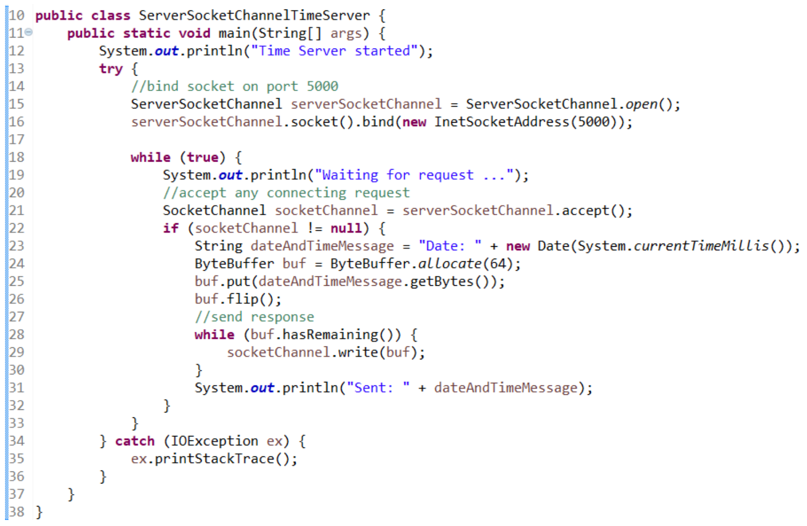
Writing a Blocking TCP Client

- Steps to implement blocking TCP client
  1. Creating a New Socket Channel
  2. Configuring Blocking Mechanisms
  3. Setting Socket Channel Options
  4. Connecting the Channel's Socket
  5. Transmitting Data over a Connection
  6. Closing the Channel

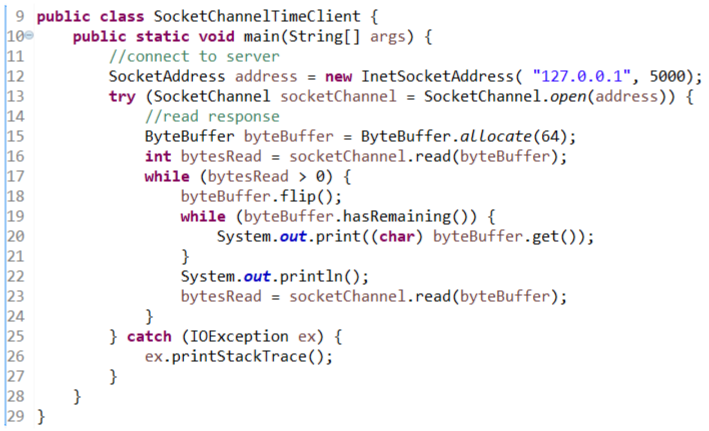


```java
public class BlockingClientDemo {
    public static void main(String[] args) {
        final String IP = "127.0.0.1"; final int DEFAULT_PORT = 5555;
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
        ByteBuffer helloBuffer = ByteBuffer.wrap("Hello !".getBytes());
        ByteBuffer randomBuffer;    CharBuffer charBuffer;
        Charset charset = Charset.defaultCharset();
        CharsetDecoder decoder = charset.newDecoder();
        //create a new socket channel
        try (SocketChannel socketChannel = SocketChannel.open()) {
            //continue if it was successfully created
            if (socketChannel.isOpen()) {
                socketChannel.configureBlocking(true);//set the blocking mode
                //set some options
                socketChannel.setOption(StandardSocketOptions.SO_RCVBUF, 128 * 1024);
                socketChannel.setOption(StandardSocketOptions.SO_SNDBUF, 128 * 1024);
                socketChannel.setOption(StandardSocketOptions.SO_KEEPALIVE, true);
                socketChannel.setOption(StandardSocketOptions.SO_LINGER, 5);
                //connect this channel's socket
                socketChannel.connect(new InetSocketAddress(IP, DEFAULT_PORT));
                //check if the connection was successfully accomplished
                if (socketChannel.isConnected()) {
                    //transmitting data
                    socketChannel.write(helloBuffer);
                    while (socketChannel.read(buffer) != -1) {
                        buffer.flip();
                        charBuffer = decoder.decode(buffer);
                        System.out.println(charBuffer.toString());
                        if (buffer.hasRemaining()) {buffer.compact();}
                        else {buffer.clear();}
                        int r = new Random().nextInt(100);
                        if (r == 50) {
                            System.out.println("50 was generated! Close the socket channel!");
                            break;
                        } else {
                            randomBuffer = ByteBuffer.wrap("Random number:".concat(String.valueOf(r)).getBytes());
                            socketChannel.write(randomBuffer);
                        }
                    }
                } else {
                    System.out.println("The connection cannot be established!");
                }
```

Writing a Non-Blocking TCP Client/Server Application



```java
public class ServerSocketChannelTimeServer {
    public static void main(String[] args) {
        System.out.println("Time Server started");
        try {
            //bind socket on port 5000
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            serverSocketChannel.socket().bind(new InetSocketAddress(5000));

            while (true) {
                System.out.println("Waiting for request ...");
                //accept any connecting request
                SocketChannel socketChannel = serverSocketChannel.accept();
                if (socketChannel != null) {
                    String dateAndTimeMessage = "Date: " + new Date(System.currentTimeMillis());
                    ByteBuffer buf = ByteBuffer.allocate(64);
                    buf.put(dateAndTimeMessage.getBytes());
                    buf.flip();
                    //send response
                    while (buf.hasRemaining()) {
                        socketChannel.write(buf);
                    }
                    System.out.println("Sent: " + dateAndTimeMessage);
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```



```
public class SocketChannelTimeClient {
    public static void main(String[] args) {
        //connect to server
        SocketAddress address = new InetSocketAddress( "127.0.0.1", 5000);
        try (SocketChannel socketChannel = SocketChannel.open(address)) {
            //read response
            ByteBuffer byteBuffer = ByteBuffer.allocate(64);
            int bytesRead = socketChannel.read(byteBuffer);
            while (bytesRead > 0) {
                byteBuffer.flip();
                while (byteBuffer.hasRemaining()) {
                    System.out.print((char) byteBuffer.get());
                }
                System.out.println();
                bytesRead = socketChannel.read(byteBuffer);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

69

70